Preprint
UCRL-JC-149277

Efficient Computation of the Topology of Level Sets

V. Pascucci, K. Cole-McLaughlin

This article was submitted to
Institute for Electrical and Electronics Engineers Visualization 2002,
Boston, Massachusetts, October 27 – November 1, 2002

July 19, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

Efficient Computation of the Topology of Level Sets*

V. Pascucci K. Cole-McLaughlin
Center of Applied Scientific Computing
Lawrence Livermore National Laboratory

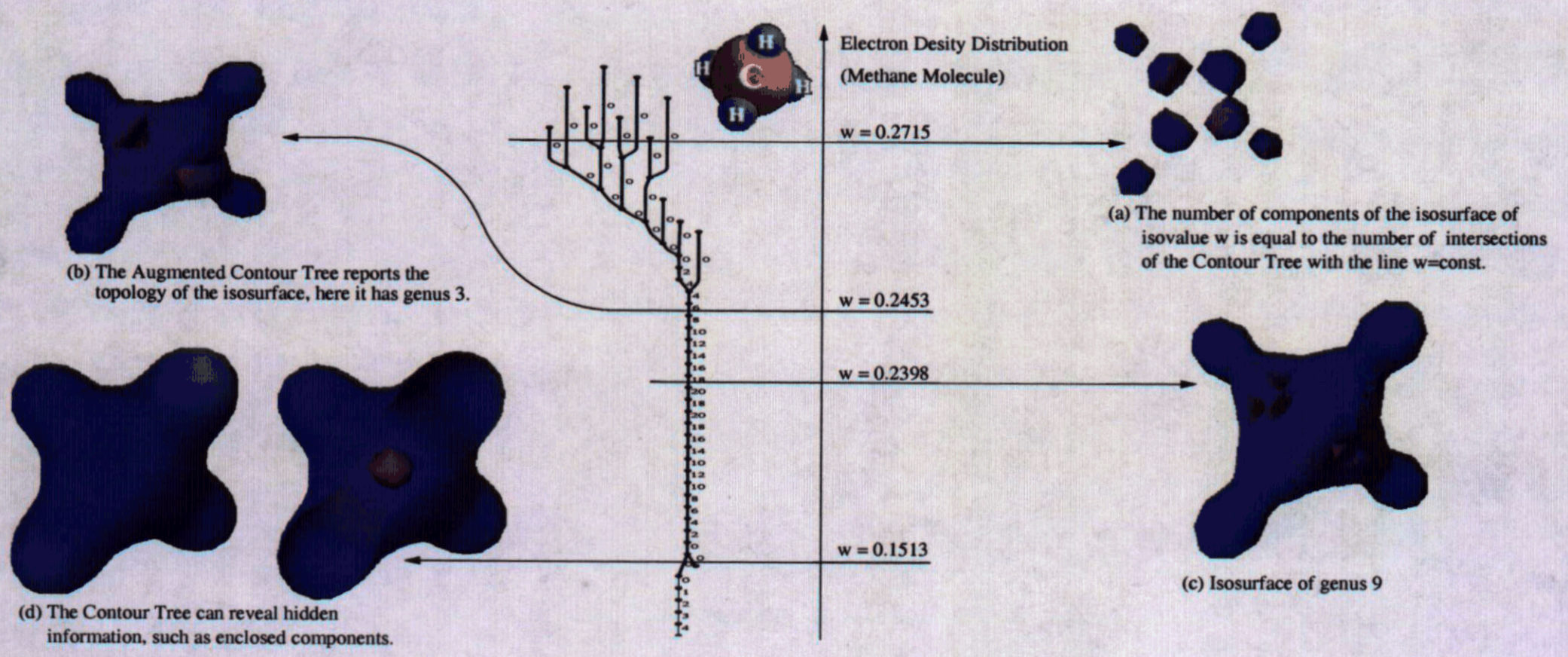

Figure 1: Augmented Contour Tree of the ectron density distribution of a methane nolecule, shown with four isosurfaces (level sets). Each arc of the Augmented Contour Tree is labeled by the second Betti number (equal to twice the number of handles of the surface) of the corresponding isosurface. The four isosurfaces are computed for isovalues $w = 0.2715$ (a), $w = 0.2453$ (b), $w = 0.2389$ (c) and $w = 0.1513$ (d). Contour (d) is shown in two viwes. The first (standard) view shows only the outer component of the isosurface. The second clipped view shows the second component in the interior, which presence is reviled by the second intersection of the horizzontal line $w = 0.1513$ with the contour tree.

Abstract

This paper introduces two efficient algorithms that compute the Contour Tree of a 3D scalar field $\mathcal{F}$ and its augmented version with the Betti numbers of each isosurface. The Contour Tree is a fundamental data structure in scientific visualization that is used to pre-process the domain mesh to allow optimal computation of isosurfaces with minimal storage overhead. The Contour Tree can be also used to build user interfaces reporting the complete topological characterization of a scalar field, as shown in Figure 1.

In the first part of the paper we present a new scheme that augments the Contour Tree with the Betti numbers of each isocontour in linear time. We show how to extend the scheme introduced in [3] with the Betti number computation without increasing its complexity. Thus we improve on the time complexity from our previous approach [8] from $O(m \log m)$ to $O(n \log n + m)$, where m is the number of tetrahedra and n is the number of vertices in the domain of $\mathcal{F}$.

In the second part of the paper we introduce a new divide and conquer algorithm that computes the Augmented Contour Tree for scalar fields defined on rectilinear grids. The central part of the scheme computes the output contour tree by merging two intermediate contour trees and is independent of the interpolant. In this way we confine any knowledge regarding a specific interpolant to an oracle that computes the tree for a single cell. We have implemented this oracle for the trilinear interpolant and plan to replace it with higher order interpolants when needed. The complexity of the scheme is $O(n + t \log n)$, where t is the number of critical points of $\mathcal{F}$. This allows for the first time to compute the Contour Tree in linear time in many practical cases when $t = O(n^{1-\epsilon})$.

We report the running times for a parallel implementation of our algorithm, showing good scalability with the number of processors.

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

1 Introduction

Scalar fields are used to represent data in different application areas like geographic information systems, medical imaging or scientific visualization.

One fundamental visualization technique for scalar fields is the display of level sets: that is, sets of points of equal scalar value. For example in terrain models isolines are used to highlight regions of equal elevation. In medical CT scans an isosurface can be used to show and reconstruct the separation between bones and soft tissues.

In molecular modeling the representation of the interaction en-

ergy between two molecules is a scalar field defined over a six dimensional configuration space. The six dimensions are the three translational and the three rotational degrees of freedom of the relative positions of the two molecules. The level sets of the field represent all the configurations which are energetically equivalent.

The domain of a scalar field is typically a geometric mesh and the field is provided by associating each vertex in the mesh with a sampled scalar value. If the mesh is a simplicial complex then a piecewise linear function is naturally defined by interpolating linearly, within each simplex, the scalar values at the vertices. If the mesh is a rectilinear grid then a piecewise trilinear function is naturally defined by interpolating, within each cell, the scalar values at the vertices.

The Contour Tree is a data structure that represents the relations between the connected components of the level sets in a scalar field. Two connected components that merge together (as one continuously changes the isovalue) are represented as two arcs that join at a node of the tree. The pre-computation of the Contour Tree allows one to collect structural information relative to the isocontours of the field. This can be used for example to speed up the computation of isosurfaces by computing seed sets over the contour tree data structure as in [11]. The display [1] of the Contour Tree provides the user with direct insight into the topology of the field and reduces the user interaction time necessary to "understand" the structure of the data. Figure 1 shows an example of how information can be extracted from the Contour Tree display.

The first efficient technique for Contour Tree computation in 2D was introduced by de Berg and van Kreveld in [5]. The algorithm proposed has $O(n \log n)$ complexity. A simplified version, with same complexity in 2D and $O(m^2)$ complexity in higher dimensions, was proposed by van Kreveld et al. in [11]. This new approach is also used as a preprocessing step for an optimal isocontouring algorithm. It computes a small seed set from which any contour can be tracked in optimal running time. The approach has been improved by Tarasov and Vyalyi [10] achieving $O(m \log m)$ complexity in the 3D case by a three pass mechanism that allows one to resolve the different types of criticalities. Recently Carr, Snoeyink and Axen [3] presented and elegant extension to any dimension based on a two pass scheme that builds a join-tree and a split-tree that are merged into a unique Contour Tree. The approach achieves $O(m + n \log n)$ time complexity.

One fundamental limitation of the basic Contour Tree is the lack of additional information regarding the topology of the contours. In high pressure chemical simulations [9] hydrogen bonds between the atoms cannot be represented in a traditional way, but can be characterized by isosurfaces of potential fields. The Contour Tree provides important information regarding the clustering of atoms into molecules but fails to discriminate between linear chains and closed rings (or more complex structures) which have different physical behaviors. In [8] we introduced the first efficient algorithm for the computation of the Betti numbers of all the level sets of a scalar field in $O(m \log m)$ time.

The first part of this paper introduces an extension of the algorithm in [3] that allows one to add the Betti numbers of each contour while maintaining the simplicity of the scheme and the efficient $O(m + n \log n)$ time complexity.

The second part of this paper introduces a new divide and conquer scheme for the computation of the Contour Tree. The basic idea is to compute merge/split-trees by combining recursively the same trees computed for two halves of the mesh. This approach allows one to achieve better modularity by confining any knowledge of a specific interpolant to an oracle that computes the tree for a single cell (in the appendix we report the oracle for the trilinear interpolant). In our analysis of the scheme for the case of rectilinear grids ($m = \Theta(n)$) we show a time complexity of $O(n + t \log n)$, where t is the number of critical points in the field.

The algorithm is also easy to parallelize. We report running times for a parallel implementation, showing good scalability with the number of processors.

2 The Contour Tree

Consider a scalar field $\mathcal{F}$ defined as a pair $(f, \mathcal{M})$, where f is a real valued function and $\mathcal{M}$ is the domain of f. In the following two sections the domain $\mathcal{M}$ is assumed to be a simplicial complex with n vertices and m cells. In Section 5 the domain $\mathcal{M}$ is assumed to be a rectilinear grid. Within each simplex of $\mathcal{M}$ the function f is the linear interpolation of its values at the vertices (trilinear for grid cells). In other words the field $\mathcal{F}$ is completely defined by the mesh $\mathcal{M} = \{v_1, \ldots, v_n\}$ and the set of scalar values $\{f_1, \ldots, f_n\}$ where $f_i = f(v_i)$. Since $\mathcal{M}$ is connected (or processed one connected component at a time) the range of f is a simple closed interval $r = [f_{min}, f_{max}]$ where $f_{min} = \min\{f_1, \ldots, f_n\}$ and $f_{max} = \max\{f_1, \ldots, f_n\}$.

For simplicity of presentation $\mathcal{M}$ is also assumed to be homeomorphic to a 3-ball. One fundamental way to study the field $\mathcal{F}$ is to extract its level sets. For a given scalar x the level set $L(x)$ is defined as the inverse image of x onto $\mathcal{M}$ through f:

$$L(x) = f^{-1}(x).$$

We call each connected component of the level set $L(x)$ a *contour*. One aspect that is well understood in Morse theory [7] is the evolution of the homology classes of the contours of $\mathcal{F}$ while x changes continuously in r. The points at which the topology of a contour changes are called critical points and the corresponding function values are called critical values. The critical points are usually assumed to be isolated. This assumption is not satisfied in general but can be enforced by small (symbolic) perturbations of the function values $\{f_1, \ldots, f_n\}$ as discussed in Section 3.

Here this perturbation procedure is weakened by simply assuming that the function values $\{f_1, \ldots, f_n\}$ are sorted from the smallest to the largest so that $i < j \Rightarrow f_i \leq f_j$. This can be enforced with an $O(n \log n)$ preprocessing step. In the following the order of the f_i is used to resolve non-isolated criticalities.

We follow the notation of [3] and define the Contour Tree (CT) as a tree whose vertices are associated with a function value f_i and whose connectivity represents the relation among the contours of $\mathcal{F}$ as follows.

- Each leaf of CT represents a local extremum where a contour is created or destroyed, for continuous changes of x. The function value of the extremum is associated with the leaf node of CT.
- Each interior vertex of CT represents the merging and/or splitting of two or more contours for continuous changes of x. The function value at which a split/merge occurs is associated with the node.
- Each arc of CT represents a contour that remains isolated for values of x ranging between the function values associated with the end nodes of the arc.

Figure 2 shows a 2D scalar field with the associated Contour Tree. Note that the Contour Tree is not a complete Morse graph of $\mathcal{F}$ since the topological changes of a single contour are not recorded. A more intuitive way to characterize the Contour Tree is the following informal definition:

> The *Contour Tree* of $\mathcal{F}$ is the graph obtained by continuous contraction of each contour of $\mathcal{F}$ to a single point. Adjacent contours are contracted to adjacent points. Distinct contours are contracted to distinct points.

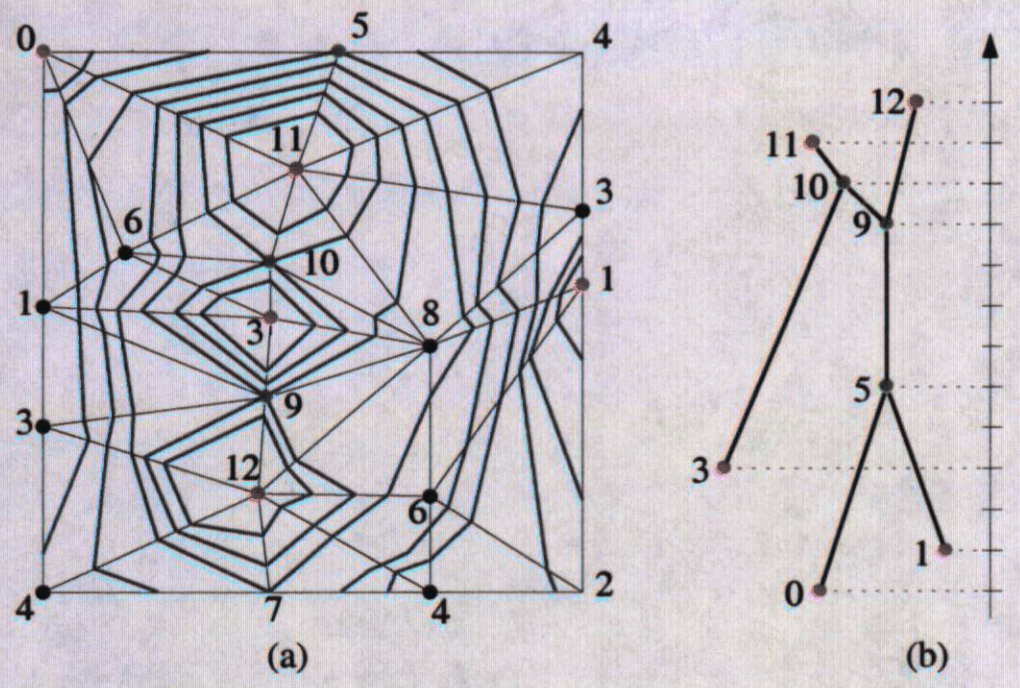

Figure 2: (a) 2D scalar field (terrain) represented as a tringulation with elevation values associted with each vertes. The critical points are marked with colored disks: maxima in red, saddles in green and minima in purple. A set of representative level sets (isolines) are drawn in blue. (b) Corresponding contour tree.

In the following we show in 3D how the Contour Tree can be efficiently computed and augmented with the complete topological inform ation of each contour. In particular we associate each arc of the tree with the Betti numbers of the corresponding contour.

3 Contour Tree Computation

This section summarizes the main result of [3], which is an elegant and efficient algorithm for the computation of the Contour Tree in any dimension. We refer to [3] for a formal proof of the correctness of the scheme.

The algorithm is divided into three stages: (i) sorting of the vertices in the field, (ii) computing the Join Tree (JT) and Split Tree (ST), and (iii) merging the JT with the ST to build the CT.

Sorting vertices. The vertices of the mesh are ordered by increasing function value in $O(n \log n)$ time using any standard sorting technique. It is important to remark that the remainder of the algorithm relies on the assumption that there are no two vertices with the same function value. Typical input fields do not satisfy this assumption, therefore we impose an symbolic perturbation of the function values by replacing the test $f(v_a) \stackrel{?}{<} f(v_b)$ with the test $a \stackrel{?}{<} b$. After the sorting this is a legitimate operation which resolves consistently the ties when $f(v_a) = f(v_b)$ while returning the same result as the original test in the other cases. In the following we also use the symbol i for the node of CT, JT or ST that corresponds to v_i.

Computing the JT and the ST. The computation of the JT and of the ST is performed in two sweeps trough the data in forward and reverse vertex order. The JT is built incrementally with a tree data-structure supporting the obvious functions NewTree(), AddNode(XT, i) and AddArc(XT, i, j). Implicitly the JT tracks the history of the UNION operations of a UNION-FIND data-structure over the set of vertices in the mesh with respectively increasing and decreasing function value. NewSet(UF, i) creates the new set $\{i\}$, with reference node i. If k belongs to the set i then Find(UF, k) returns i in constant time. Union(UF, i, j) redirects the pointers of all the elements in j to point to i, if i has larger cardinality than j (vice versa if $|i| < |j|$).

The Boolean function IsMin($\mathcal{F}, v_i$) returns true if v_i is a local minimum in $\mathcal{F}$.

```
JoinTree(vertices, edges)
 1  JT= NewTree()
 2  UF= NewUF()
 3  for i = 0 to n − 1 do:
 4    AddNode(JT, i)
 5    if IsMin(F, v_i) then NewSet(UF, i)
 6    for each edge v_i v_j with j < i do:
 7      i' ← Find(UF, i)
 8      j' ← Find(UF, j)
 9      if j' ≠ i' then AddArc(JT, i', j')
10      Union(UF, i', j')
12  return JT
```

Each vertex v_i is associated with two lists UpAdj, of incident edges (v_i, v_j) with $j > i$, and DownAdj of incident edges (v_i, v_j) with $j < i$. In this way IsMin($\mathcal{F}, v_i$) can test in constant time if i is a minimum (DownAdj is empty) and the loop on line 6 scans directly the elements of DownAdj.

The routine SplitTree has the same structure as JoinTree. The only differences are as follows: (i) the main loop (line 3) would scan the vertices in reverse order, (ii) the **if** statement in line 5 would test IsMax instead of IsMin and (iii) the inner loop (line 6) would consider the edges (v_i, v_j) with $j > i$. These routines are shown in [3] to have worst case time complexity of $O(m + t \log t)$

Merging the JT with the ST. In the last stage of the algorithm the JT is merged with the ST to build the CT. The upper leaves of the JT and the lower leaves of the ST are successively removed from both trees and added to the CT. Consequently the data-structure representing the JT and the ST has to support the additional operations DelNode(XT, i), and Leaf(XT, i). DelNode(XT, i) removes the node i from XT while maintaining the consistency of XT by removing any arc ij and replacing any pair of arcs ij, ik with the arc jk. The Boolean function Leaf(XT, i) tests whether the node i is a leaf of XT. More specifically Leaf(JT, i) is true if the JT has no arc ij with $j < i$ and Leaf(ST, i) is true if the ST has no arc ij with $j > i$. GetAdj(XT, i) returns a vertex j if XT contains the arc ij. A queue data-structure is used to store pairs $[NodeName, TreeName]$, and is managed with the functions NewQ() (to create a queue), Get(Q) (to get a pair from the queue Q) and Put($Q, [i, XT]$) (to add a pair to Q).

```
ContourTree(JT, ST)
 1  Q ← NewQ()
 2  CT ← NewTree()
 3  for i = 0 to n − 1 do:
 4    AddNode(CT, i)
 5    if Leaf(JT, i) then Put(Q, [i, JT])
 6    if Leaf(ST, i) then Put(Q, [i, ST])
 7  while [i, XT] ← Get(Q) do:
 8    j ← GetAdj(XT, i)
 9    DelNode(ST, i)
10    DelNode(JT, i)
11    AddArc(CT, ij)
12    if Leaf(XT, j) then Put(Q, [j, XT])
13  return CT
```

One can minimize the size of the CT by deleting any node that has exactly degree two with DelNode. This reduction to a minimal

CT can be done directly during the construction of the JT and of the ST. This makes the algorithm slightly more complicated but has the advantage of reducing the size of the intermediate storage.

This last stage of the algorithm has $O(n)$ complexity. Overall the algorithm for constructing the CT has $O(m+n\log n)$ complexity, since t is never greater than n.

4 Betti Numbers Computation

This section introduces a modification to the function ContourTree that provides a more detailed characterization of the contours of a scalar field. The output generated by the modified function is the Augmented Contour Tree (ACT), as defined in [8], which has a triple $(\beta_0, \beta_1, \beta_2)$ of Betti numbers associated to each arc of the tree. The k-th Betti number β_k of a simplicial complex is the rank of its k-dimensional homology group. We restrict our attention to level sets of 3D scalar fields, which are 2-dimensional complexes. In this case only the first three Betti numbers may be non-zero. Their intuitive interpretation is as follows: β_0 is the number of connected components, β_1 is the number of independent tunnels, and β_2 is the number of voids enclosed by the surface.

Figure 3(a) shows the minimal CT for a simple scalar field that has one minimum at isovalue $x = 0$. The level set $f^{-1}(0)$ is a single contour coincident with the boundary of the mesh (on the bottom left). As the isovalue is continuously increased, the level set splits into four contours at isovalue $x = 2$ (on the middle left). Each contour shrinks to a single point and disappears at the maximum isovalue $x = 4$ (on the top left). Figure 3(b) shows the minimal ACT for the same scalar field. The added information allows the user to observe that the level set at the minimum is topologically a sphere ($\beta_0 = 1, \beta_1 = 0, \beta_1 = 1$) which turns into a toroidal contour ($\beta_0 = 1, \beta_1 = 2, \beta_1 = 1$) at isovalue $x = 1.2$. The toroidal contour then splits into four components each being a topological sphere.

In general the ACT has the same structure of the CT since it has the same nodes of degree not equal to two (extrema and merge/split points) and the same connectivity among them. The main difference between the two trees is that the CT, in its minimal form, has no nodes of degree two. In contrast the ACT requires degree two nodes at the isovalues where a contour changes its topology without splitting or merging. Because of these added nodes, each arc of the ACT is associated with a family of contours that are homologically equivalent and hence qualified by the same set of Betti numbers. Moreover the contours associated with an arc contain no critical points and the Betti numbers are restricted as follows: (i) β_0 is always 1, (ii) β_2 is 0 for surfaces with boundary (open) and is 1 for surfaces without boundary (closed). Once β_0 and β_2 are determined we can compute the value of β_1 using its relationship with the Euler characteristic χ:

$$\chi = \beta_0 - \beta_1 + \beta_2. \tag{1}$$

Given a triangulated surface, the Euler number χ is defined as the number of vertices minus the number of edges plus the number of faces. In addition to computing the Euler number, for each contour we count the number of boundary edges (be). In this way we can determine β_2 by checking if $be > 0$ and then use (1) to compute $\beta_1 = \beta_0 + \beta_2 - \chi$.

In a preliminary stage we compute, for each vertex v, the information necessary to determine the difference between the Euler number of the level set $L(f(v) + \epsilon)$ and the Euler number of the level set $L(f(v) - \epsilon)$ where $\epsilon > 0$ is an arbitrarily small number (remember that $f(v) = f(w)$ implies $v = w$). Figure 4 shows two such level sets for a 2D scalar field. The vertices with function value greater than $f(v)$ are marked $\oplus$ and the vertices with function value smaller than $f(v)$ are marked $\ominus$. Any simplex containing

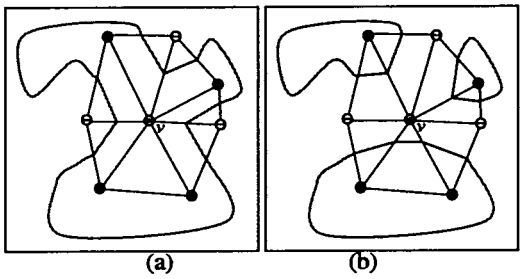

Figure 4: Comparison between two level sets (isolines in gray) of a 2D scalar field. (a) shows an isoline of isovalue $f(v) - \epsilon$. (b) shows an isoline of isovalue $f(v) + \epsilon$. The difference between combinatorial structure of the two isolines is confined within the star of simplices incident to v.

both vertices of type $\oplus$ and vertices of type $\ominus$ give the same contribution to the Euler numbers of the two contours and hence are not considered. The only simplices that are relevant are those containing v and only vertices of type $\ominus$ or those containing v and only vertices of type $\oplus$. We call the lower star of v the set of simplices of the first type $(v, \ominus, \ldots, \ominus)$ and the upper star the set of simplices of the second type $(v, \oplus, \ldots, \oplus)$. For both stars we compute the respective Euler numbers LS and US (number of vertices minus number of edges plus number of triangles minus number of tetrahedra). We also count the difference Δbe between the boundary edges of $L(f(v) - \epsilon)$ and $L(f(v) + \epsilon)$. This is summarized in the following algorithm.

LUStars($vertices, edges, triangles, tetrahedra$)
1 **for** $i = 0$ **to** $n - 1$ **do:**
2 $\quad LS_i = US_i = 1$
3 $\quad \Delta be_i = 0$
4 **for each edge** (v_i, v_j) **with** $i < j$ **do :**
5 $\quad LS_j \leftarrow LS_j - 1$
6 $\quad US_i \leftarrow US_i - 1$
7 **for each triangle** (v_i, v_j, v_k) **with** $i < j < k$ **do :**
8 $\quad LS_k \leftarrow LS_k + 1$
9 $\quad US_i \leftarrow US_i + 1$
10 $\quad$ **if** (v_i, v_j, v_k) **is a boundary triangle then:**
11 $\quad\quad \Delta be_k \leftarrow \Delta be_k - 1$
12 $\quad\quad \Delta be_i \leftarrow \Delta be_i + 1$
11 **for each tetrahedron** (v_i, v_j, v_k, v_l) **with** $i < j < k < l$ **do :**
12 $\quad LS_l \leftarrow LS_l - 1$
13 $\quad US_i \leftarrow US_i - 1$
14 **return**$(LS, US, \Delta be)$

From a CT that contains all the nodes we build the corresponding ACT. We call χ_{ij} the Euler number of the contour associated with the arc ij of the CT. For any fixed i the summation $\sum \chi_{ij}$, with $j < i$, is the sum of the Euler numbers of the contours of $L(f(v_i) - \epsilon)$ which intersect the star of v_i. Similarly we denote by be_{ij} the number of boundary edges of the contour associated with the arc ij.

We consider, at a generic node i, the relation between LS_i, US_i and the Euler numbers of the contours associated with the arcs incident to i. In particular each edge, triangle and tetrahedron in the lower star of v_i produces one vertex, edge and face, respectively, in some contour of $L(f(v_i) - \epsilon)$. In the same way each edge, triangle and tetrahedron in the upper star of v_i produces one vertex, edge and face, respectively, in some contour of $L(f(v_i) + \epsilon)$. Since these two terms are the only difference between the Euler numbers of $L(f(v_i) - \epsilon)$ and of $L(f(v_i) + \epsilon)$ we can write:

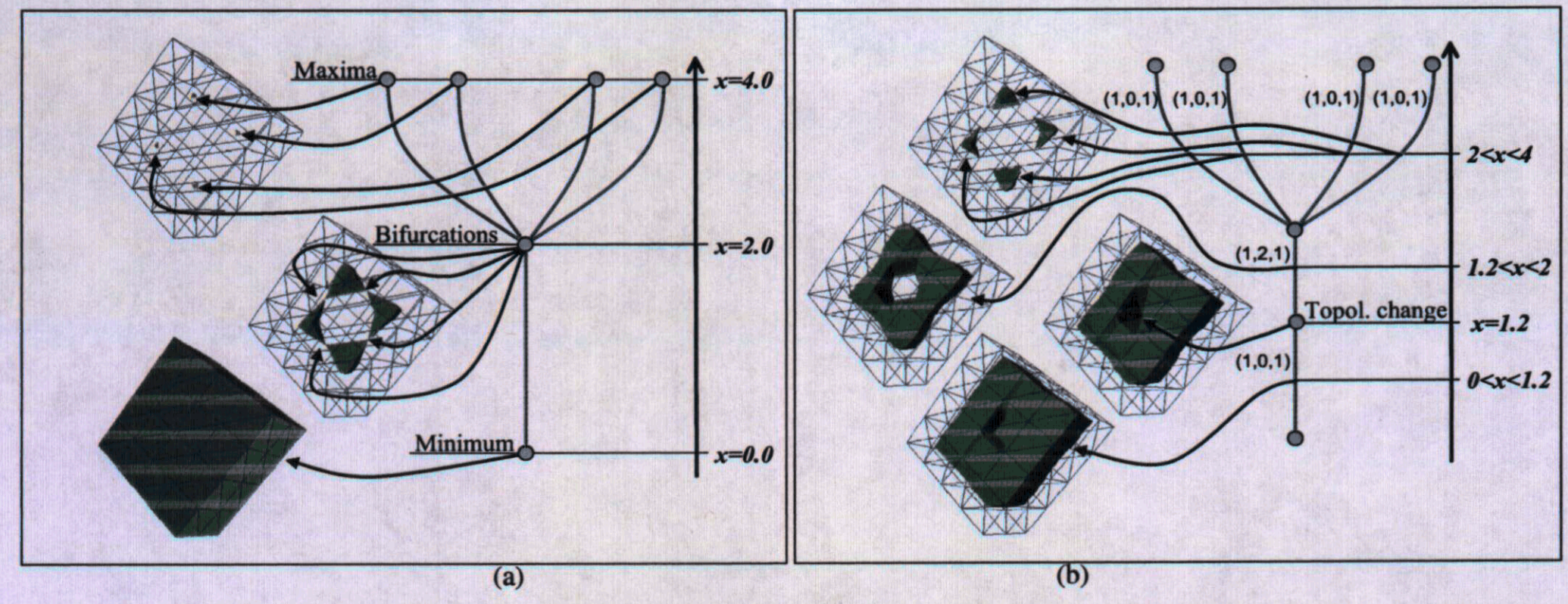

Figure 3: (a) Information provided by the standard CT for a simple scalar field. (b) The added information provided by the ACT provides a better understanding of the structure of each contour.

$$\sum_{ij|j<i} \chi_{ij} + LS_i = \sum_{ij|j>i} \chi_{ij} + US_i$$

Overall we have a set of n linear equations, one for each node of the ACT, with $n-1$ unknowns χ_{ij}. To solve this system we define n artificial variables χ_i that are initially set to zero. In this way one can rewrite the linear equations as follows:

$$\chi_i + \sum_{ij|j<i} \chi_{ij} + LS_i = \sum_{ij|j>i} \chi_{ij} + US_i \qquad (2)$$

A similar argument holds for the count of the boundary edges be_{ij} of each contour. We define an array of auxiliary variables be_i that are initially set to zero and satisfy the following equations:

$$be_i + \sum_{ij|j<i} be_{ij} + \Delta be_i = \sum_{ij|j>i} be_{ij} \qquad (3)$$

We solve the systems of linear equations defined by (2) and (3) with the procedure AugmentedContourTree, which incrementally moves an arc ij from the CT to the ACT each time the corresponding value of χ_{ij} can be determined (the function Degree(XT, v) returns the degree of the node v in XT):

AugmentedContourTree($CT-with-all-nodes$)
1 $Q \leftarrow$ NewQ()
2 $ACT \leftarrow$ NewTree()
3 **for** $i = 0$ **to** $n-1$ **do:**
4 $\quad \chi_i \leftarrow 0$
5 $\quad be_i \leftarrow 0$
6 $\quad$ AddNode(ACT, i)
7 $\quad$ **if** Degree$(CT, i) = 1$ **then** Put(Q, i)
8 **while** $i \leftarrow$ Get(Q) **do:**
9 $\quad j \leftarrow$ GetAdj(CT, i)
10 $\quad$ AddArc(ACT, i, j)
11 $\quad$ **if** $i < j$ **then** $\delta \leftarrow +1$ **else** $\delta \leftarrow -1$
12 $\quad \chi_{ij} \leftarrow \delta(\chi_i - US_j + LS_j)$
13 $\quad be_{ij} \leftarrow \delta(be_i + \Delta be_i)$
14 $\quad \chi_j \leftarrow \chi_j + \delta \cdot \chi_{ij}$
15 $\quad be_j \leftarrow be_j + \delta \cdot be_{ij}$
16 $\quad$ DelNode(CT, i)
17 $\quad$ **if** Degree$(CT, j) = 1$ **then** Put(Q, j)
18 **return** ACT

Note that the while loop in line 8 has the same structure of the while loop in line 7 of ContourTree (see page 3). Therefore one can compute directly the Euler numbers χ_{ij} and merge the JT with the ST in the same loop. The Betti numbers can also be added at the same time. For completeness we report the function that computes the Betti numbers as a post-processing.

BettiNumbers(ACT)
1 **for each arc** ij **of** ACT **do:**
2 $\quad \beta_{0,ij} \leftarrow \beta_{2,ij} \leftarrow 1$
3 $\quad$ **if** $be_{ij} \neq 0$ **then** $\beta_{2,ij} \leftarrow 0$
4 $\quad \beta_1 \leftarrow \beta_0 + \beta_2 - \chi_{ij}$

ACT Reduction. The following function, Reduce, removes all of the non-critical points from the ACT in order to reduce it to its minimal form. The test is based on the critical point theorem in [2] and can detect the critical points in constant time once the arrays LS and US have been computed. Note that this removal of non-critical points can be done during the computation of the ACT reducing the necessary intermediate storage.

Reduce(ACT)
1 **for** $i = 0$ **to** $n-1$ **do:**
2 $\quad$ **if** $LS_i = US_i = 0$
3 $\quad\quad$ DelNode(ACT, i)

Correctness. The correctness of the routines LUStars and BettiNumbers derives directly from the definitions of the parameters computed. To prove the correctness of AugmentedContourTree we show that there are two invariants which remain true at each iteration. The invariants are the systems of equations (2) and (3). Initially both systems are true by definition, since all the χ_i and the be_i are set to zero. We focus only on equations (2) since the same argument holds for (3).

At each iteration of the while loop (line 8) a leaf i is selected from the CT together with its incident arc ij. Therefore the ith equation of (2) has only one unknown, χ_{ij}. χ_{ij} is computed with

the explicit formula $\chi_{ij} = LS_i - US_i + \chi_i$, if $j > i$, or with the explicit formula $\chi_{ij} = US_i - LS_i - \chi_i$, if $j < i$. The node i and the arc ij are then removed from CT invalidating the jth equation of (2) since the term χ_{ij} is no longer present. We restore its correctness by adding the value of χ_{ij} to χ_j, if $j > i$ (or subtracting if $j < i$). Thus, after each iteration the CT is reduced by an arc, while the systems (2) and (3) remain true.

At the end of the loop the tree CT has no arcs and all the terms χ_{ij} and be_{ij} are computed. □

Complexity. The complexity of the procedure LUStars is $O(m)$ while the complexity of AugmentedContourTree and BettiNumbers is $O(n)$. Overall the computation of the ACT with the Betti numbers remains $O(m + n \log n)$. This is an improvement over the previous $O(m \log m)$ achieved in [8] since m can be as big as $O(n^2)$.

5 Divide and Conquer for Rectilinear Grids

This section introduces a new way to compute the JT and the ST using a divide and conquer strategy. We restrict our analysis to the case of scalar fields $\mathcal{F} = (f, \mathcal{M})$ where $\mathcal{M}$ is a rectilinear mesh of dimensions $n_x \times n_y \times n_z$. This is the type of mesh that typically has the largest number of vertices (i.e. the type used in the largest simulations or generated by high resolution MRI/CT scanning devices). In this case the function f is defined within each cell as the trilinear interpolation of the field values at the eight vertices. In this framework we cannot use the algorithm ContourTree since it assumes properties that are specific to a piecewise linear interpolant. For example, the trilinear interpolant admits critical points in the interior of a cell, a condition not allowed by ContourTree. Triangulating the cells of the grid is usually not an option for large data-sets especially because the same topology cannot be reproduced in general unless several more vertices are added to each cell of the mesh.

Our approach overcomes this problem by assuming an oracle OracleJT$(\mathcal{F}, \mathcal{M})$ that returns the JT of $\mathcal{F}$ if $\mathcal{M}$ is a single cell. We have implemented such an oracle for the trilinear interpolant on a cube (see Appendix). To extend the scheme to data-sets with other types of interpolants, for example a triquadratic interpolant, requires only to replace the function OracleJT. OracleST$(\mathcal{F}, \mathcal{M})$ is simply OracleJT$(-\mathcal{F}, \mathcal{M})$.

Recursive algorithm. The algorithm has the same structure of a merge sort scheme with the added feature that non-critical vertices are removed as soon as possible. This removal provides an output sensitive character to the algorithm that improves both its time complexity and its space complexity.

```
RecursiveJT(F, M)
 1 if Dimensions(M) = (2,2,2) then
 2   return OracleJT(F, M)
 3 [M1, M2] ← Split (M)
 4 JT1 ← RecursiveJT (F, M1)
 5 JT2 ← RecursiveJT (F, M2)
 6 JT ← MergeJT(JT1, JT2)
 7 return Reduce(JT)
```

The function Split $(\mathcal{M})$ divides in constant time the domain of the mesh into two approximately equal meshes $\mathcal{M}_1$ and $\mathcal{M}_2$. In particular if $\mathcal{M}$ has size (n_x, n_y, n_z), with $n_x \geq n_y \geq n_z$, then $\mathcal{M}_1$ has size (n'_x, n_y, n_z) and $\mathcal{M}_2$ has size (n''_x, n_y, n_z), where $n'_x = \lceil n_x/2 \rceil$ and $n''_x = n_x + 1 - n'_x$.

Tree merging. The routine MergeJT below combines the join trees of the two halves of the mesh using a UnionFind data-structure in the same way the routine JoinTree computes the global JT from the edges of the mesh. Two key differences need to be highlighted:

- MergeJT sorts the input nodes in linear time since JT_1 and JT_2 have their nodes already sorted. In particular one linear scan through the input trees sorts the nodes and at the same time merges the duplicate nodes, which correspond to vertices on the surface $\mathcal{M}_1 \cap \mathcal{M}_2$. This task is performed by MergeNodesSorted which also returns the total number of distinct nodes.

- MergeJT copies verbatim into JT the independent portions of JT_1 and of JT_2. This is done in linear time. The UnionFind data-structure is used starting at the nodes that correspond to local minima of the scalar field restricted to $\mathcal{M}_1 \cap \mathcal{M}_2$ ($\mathcal{F}|_{\mathcal{M}_1 \cap \mathcal{M}_2}$). The test for minima is performed by IsMin in constant time.

```
MergeJT(JT1, JT2)
 1 JT= NewTree()
 2 UF = NewUF()
 3 k ← MergeNodesSorted(JT1, JT2))
 4 for each node i = 0 to k − 1 do:
 5   AddNode(JT, i)
 6   if IsMin(F|M1∩M2, i) then NewSet(UF, i)
 7   for each edge vivj with j < i do:
 8     i' ← Find(UF, i)
 9     j' ← Find(UF, j)
10     if j' ≠ i' then AddArc(JT, i', j')
11     Union(UF, i', j')
12 return JT
```

Let n be the number of vertices of $\mathcal{M}_1$ and $\mathcal{M}_2$, k be the number of nodes of JT_1, JT_2 and t be the number of the minima of $\mathcal{F}|_{\mathcal{M}_1 \cap \mathcal{M}_2}$. The complexity of MergeJT is $O(n^{2/3} + k + t \log t)$. Since $t = O(n^{2/3})$ we can rewrite the complexity as $O(n^{2/3} \log n + k)$.

ACT **Reduction.** As shown in section 4, Reduce can test if a point i is non-critical simply by looking at LS_i and US_i. In this context IsRegular performs the same combinatorial test modified for the interpolant used by OracleJT. Note that the last call to Reduce should be modified to not check IsInterior, so that all of the non-critical points are removed. Otherwise non-critical points on the boundary of the mesh would remain in ACT.

```
Reduce(ACT)
 1 for i = 0 to n do:
 2   if IsInterior(i) and IsRegular(i)
 3     DelNode(ACT, i)
```

Complexity. To determine the complexity of RecursiveJT we analyze separately the cost of dealing with the interior critical points and the cost of dealing with the boundaries that are artificially introduce by the subdivision process and removed by MergeJT.

We assume that n is the number of cells of $\mathcal{M}$ and that Split partitions $\mathcal{M}$ into two equal halves of size $n/2$. Therefore the number of levels in the recursion tree of RecursiveJT is $\log n$.

The function OracleJT, which takes constant time, is invoked exactly n times (once per cell), accounting for a $\Theta(n)$ time complexity.

As the sub-meshes are merged together boundary points become interior points. In particular every point is processed by **Reduce** in constant time. Moreover any point that fails the test **IsRegular** is also processed in constant time by **MergeJT** at every level of the recursion. If $\mathcal{F}$ has t critical points we spend $O(n + t \log n)$ time to find and process them.

To analyze the cost of dealing with the boundaries we apply the master theorem of recursive functions reported on page 62 of [4]. The theorem allows one to determine the complexity of a function $T(n)$ on the basis of the recurrence formula $T(n) = 2T(n/2) + f(n)$ and the complexity of the function $f(n)$. In this case $T(n)$ is the complexity of our recursive algorithm and $f(n)$ is the complexity of **MergeJT** with reference to the boundary points only (the other points have already been accounted for). As discussed earlier the highest cost in **MergeJT** is due to the Union-Find, which we have set conservatively to $O(n^{2/3} \log n)$. This means that $f(n)$ has complexity $O(n^{1-\epsilon})$ for some ϵ and hence $T(n) = \Theta(n)$. In conclusion the complexity of **RecursiveJT** is $O(n + t \log n)$. For practical cases where t is less than linear we have $t = O(n^{1-\epsilon})$ which means the overall complexity is $O(n)$.

For the case of large data-sets it is also crucial to minimize the cost of any auxiliary storage. Beyond linear storage in the size t of the output, **RecursiveJT** keeps a storage proportional to the boundary of the mesh. Overall the auxiliary storage is $O(t + n^{2/3})$.

6 Practical Results

This section reports some practical results from our implementation of the the two algorithms discussed in Sections 4 and 5. We first present an example of the Augmented Contour Tree of the scalar field obtained for a simple molecular data-set (methane) that shows surprisingly intricate topological structures. Next we compare the timings for the computation on data-sets of five different sizes.

Methane. We consider the topological analysis of a small scalar field computed by an *ab initio* simulation conditions for the methane molecule. We have computed the ACT and displayed it using the graph drawing tool **graphviz** [6]. The top portion of this graph is shown in figure 1, along with several isosurfaces, and their corresponding points in the ACT. We focus on this portion of the dataset since it is known that the simulation becomes less reliable at lower densities.

The Methane dataset, which is on a 32x32x32 rectilinear grid, is the simplest non-trivial dataset we explored. It is a nice example, since the visualization of the tree is possible by conventional means. This gives us a good way of exploring the possibilities of using the ACT as an interface for data understanding. We see from the isosurfaces (b), (c) and (d) that there is useful information summarized in the ACT which is not obvious from the visualization. The isosurfaces (b) and (c) can be seen immediately to have $\beta_1 = 6$ and $\beta_1 = 18$ respectively, which implies that their respective genus $g = 3$ and $g = 9$ since $g = \beta_1/2$ for closed surfaces. In the isosurface (d) the initial visualization shows a single surface, whereas the ACT shows 2 distinct components. Only after adding a clipping plane the second component is shown to be enclosed within the first.

Performance. We have implemented in parallel the divide and conquer ACT algorithm on a shared memory platform. This is simply done by creating at each recursion two processes that compute join and split trees for each half of the mesh. The recursion become sequential as soon as the desired number of processes is reached. Table 1 summarizes running times for four data-sets of sizes scaling from thousands to millions of vertices. The speedup relative to the sequential case is reported in figure 5, compared to the ideal linear speedup (top line in the chart).

NP	HiPIP 64x64x64	Rho 128x128x128	Engine 256x256x110	Foot 125x255x176
1	1.0000	1.0000	1.0000	1.0000
2	1.9754	1.9801	1.9988	1.9993
4	3.7633	3.9168	3.9445	3.8986
8	7.4461	7.6365	7.3503	7.0672
16	13.949	15.457	14.302	12.864
32	26.465	28.460	27.132	20.797

Table 1: Performance results for four sample data-sets. The values given are the speedups achieved in computing the ACT on NP processors as compared to the case NP=1. The ideal speedup would be NP times faster.

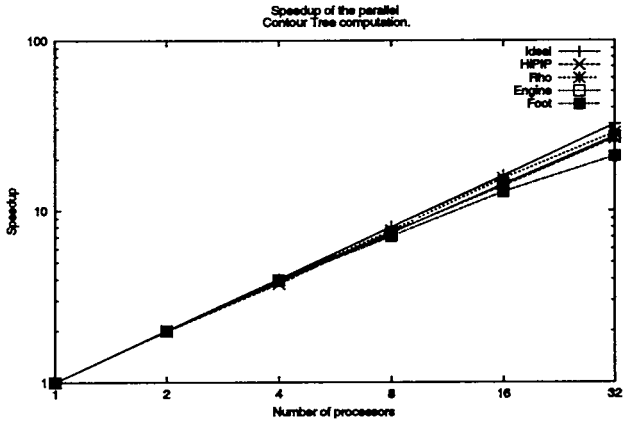

Figure 5: Practical speedups obtained in the parallel implementation for four datasets of different sizes, compared with the ideal linear speedup.

One can see that the speedup obtained in the parallel implementation scales nearly linearly. The coarse grained subdivision in our method is easily implemented in parallel. Each processor becomes responsible for a connected subregion of the mesh, and works completely independently of the other processes. The only communication necessary is for a child process to return the JT and ST that it computed to its parent.

7 Conclusions

In this paper we have introduced two schemes for the computation of the ACT for scalar fields defined on simplicial meshes and on rectilinear grids. The first scheme is an extension of the algorithm proposed in [3] with the computation of the Betti numbers.

The second contribution is a divide and conquer scheme for rectilinear grid domains. The complexity of this second scheme is improved further to $O(m + t \log n)$ where t is the number of critical points in the mesh. Moreover we demostrate good practical scalability of a simple parallel implementation of this algorithm.

The comparison between the two schemes is interesting even if they apply to different classes of inputs. In particular the divide and conquer approach seems to present several advantages especially for the processing of large data-sets. For instance, the auxiliary storage is kept as low as $O(n^{2/3} + t)$. In contrast the original scheme can have $O(n)$ auxiliary storage since the union find pro-

cessing needs to maintain auxiliary information on a set of vertices as large as the largest isosurface in the mesh.

In principle there seem to be no major problems preventing the application of the divide and conquer scheme to unstructured meshes but further investigation is necessary to verify if the same performance benefits can be guaranteed in general.

The simple task of drawing the CT has become a major problem. For data-sets that we have successfully processed we already obtain trees that current graph drawing tools cannot handle. Still we plan to work on data-sets that are orders of magnitude larger. In such cases the development of interfaces that display the CT will present a major challenge.

References

[1] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. The contour spectrum. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization 97*, pages 167–175. IEEE, November 1997.

[2] Thomas Banchoff. Critical points and curvature for emmbedded polyhedra. *Differential Geometery*, 1(1):245–256, 1967.

[3] Hamish Carr, Jack Snoeyink, and Ulrike Axen. Computing contour trees in all dimensions. *Computational Geometry Theory and Applications*, 2001. To Appear (extended abstract appeared at SODA 2000).

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] Mark de Berg and Marc J. van Kreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18(3):306–323, July 1997.

[6] John Ellson, Emden Gansner, Eleftherios Koutsofios, John Mocenigo, Stephen North, , and Gordon Woodhull. Graphviz. AT&T Research `http://www.research.att.com/~north/graphviz/`.

[7] J. Milnor. *Morse Theory*, volume 51 of *Annals of Mathematics Studies*. Princeton University Press, 1963.

[8] Valerio Pascucci. On the topology of the level sets of a scalar field. In *12th Canadian Conference on Computational Geometry*, pages 141–144, August 2001.

[9] Eric Schwegler, Giulia Galli, and Francois Gygi. Water under pressure. *Physical Review Letters*, 84(11):2429–2432, 2000.

[10] Sergey P. Tarasov and Michael N. Vyalyi. Construction of contour trees in 3d in o(n log n) steps. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 68–75, Minneapolis, June 1998. ACM.

[11] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 212–220, June 1997. Extended version. Techincal report UCRL-JC-132016 Lawrence Livermore National Laboratory.

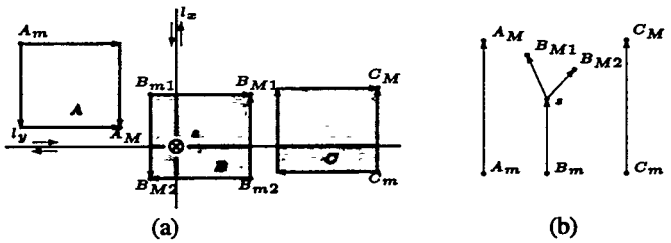

Figure 6: (a) 2D bilinear function. The saddle point s is marked with the symbol ∘. The horizontal line l_y and the vertical line l_x have constant function value and intersect at s. The orientation of the edges of the rectangles A,B,and C is along growing F. (b) Split trees of F restricted to the rectangles A, B and C. B_m is the minimum between B_{m1} and B_{m2}.

Appendix

We consider the problem of computing the merge and split trees for a cell with a trilinear interpolant. Our analysis is limited to the split tree since the join tree is computed symmetrically. We show that in the 2D case there are only two possible split trees and in 3D there are only four possible split trees. In both cases the topology of the split tree is completely determined by the number of maxima present in the cell.

Bilinear Interpolant on a Rectangle

Consider a bilinear function $F : R^2 \to R$. The analytical definition of F and its gradient ∇F is as follows:

$$\begin{aligned} F(x,y) &= axy + bx + cy + d, \\ \nabla F &= \begin{bmatrix} \partial F/\partial x \\ \partial F/\partial y \end{bmatrix} = \begin{bmatrix} ay + b \\ ax + c \end{bmatrix}, \end{aligned} \tag{4}$$

where a, b, c, d are real numbers. Since $\partial F/\partial x$, and $\partial F/\partial y$, are linear functions it is not possible to have a local maximum or minimum for finite values of x and y. Imposing $\nabla F = 0$ one finds the unique saddle point s for $x = -c/a$, and $y = -b/a$. Moreover, F is constant along the vertical line $l_x : x = -c/a$, and the horizontal line $l_y : y = -b/a$. Since $\partial F/\partial x$ is not a function of x the restriction $F|_{y=const}$ of F to any line parallel to the x axis has constant gradient. The gradient of $F|_{y=const}$ on all the lines above l_y is anti-parallel to the gradient of $F|_{y=const}$ on all the lines below l_y (see Figure 6). Similarly, l_x separates the vertical lines where $F|_{x=const}$ has upward gradient from those with downward gradient.

Fact 1 *There is exactly one saddle point s of F in the plane.*

Fact 2 *The function F is constant on the line l_x orthogonal to the x and on the line l_y orthogonal to the y axis, where l_x intersects l_y at the saddle point s of F.*

We analyze the restriction of F to axis aligned rectangles. Since F is linear along each line parallel the coordinate axis we can mark each edge of a square with respect to the direction of increasing values of F. Figure 6(a) shows the three different types of squares that one can have with respect to the orientation of their edges. A square of type A has each pair of opposite edges with parallel orientation. Therefore A cannot intersect l_x or l_y. This type of square has one maximum A_M and one minimum A_m for $F|_A$. A square of type B has both pair of opposite edges with anti-parallel orientation. Therefore B intersects both l_x and l_y. The saddle point s

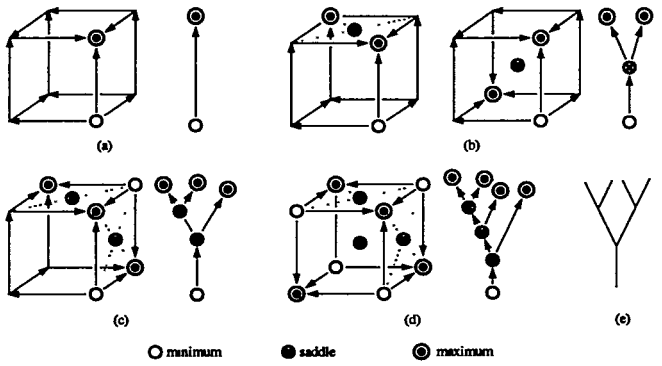

Figure 7: Possible configurations of split tree for a trilinear interpolant restricted to and axis aligned parallelepiped. One the left of each tree there are one or two examples of corresponding parallelepipeds. (a) One maximum. (b) Two maxima. (c) Three maxima. (d) Four maxima. (e) Split tree with four maxima that cannot be constructed.

must be inside B because it is at the intersection between l_x and l_y. All four vertices of B are extrema (two maxima and two minima) of $F|_B$. In the third type of square C one pair of opposite edges are parallel while the other pair are anti-parallel. Thus, C must intersect either l_x or l_y, and $F|_C$ has one maximum and one minimum.

Fact 3 *The bilinear function F restricted to an axis aligned rectangle can have only one or two maxima. The maxima can be located only at non-adjacent vertices.*

Figure 6(b) shows how the split trees of $F|_A$ and of $F|_C$ are both single lines connecting the minimum to the maximum. The split tree of $F|_B$ has one line that connects the lower minimum to the saddle s. At s the split tree of $F|_B$ bifurcates into two lines connecting s to the two maxima.

Trilinear Interpolant on a Parallelepiped

We extend our analysis to the trilinear case and show how to compute the shape of the split and merge trees for a cube on the basis of the orientation of its edges and the function value of the eventual body saddle points. The general formulation of the trilinear interpolant is:

$$F(x,y,z) = axyz + bxy + cxz + dyz + ex + gy + hz + k, \quad (5)$$

with gradient:

$$\nabla F = \begin{bmatrix} e + by + cz + ayz \\ g + bx + dz + axz \\ h + cx + dy + axy \end{bmatrix}.$$

It is easy to see that restricting (5) to any plane orthogonal to a coordinate axis (for example of equation $x = const$) yields the bilinear function of type (4). Therefore there is no local minimum or maximum of F. Solving $\nabla F = 0$ we find two critical points of coordinates:

$$x = \frac{d(ae-bc) \pm \sqrt{\Delta}}{a(bc-ae)}, y = \frac{c(ag-bd) \pm \sqrt{\Delta}}{a(bd-ag)}, z = \frac{b(ah-cd) \pm \sqrt{\Delta}}{a(cd-ah)},$$

where the term $\sqrt{\Delta}$ is either added in all expressions or subtracted in all expressions, and Δ is:

$$\Delta = (bc-ae)(bd-ag)(cd-ah).$$

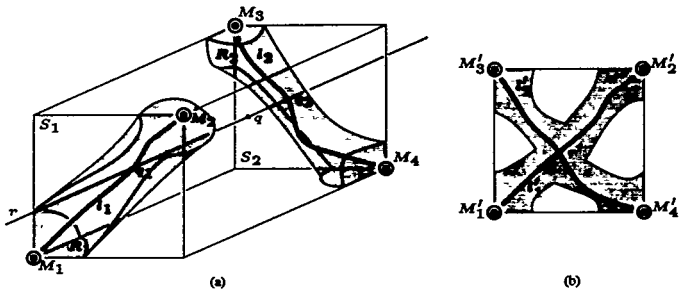

Figure 8: Impossible configurations that would be necessary to allow the construction of a split tree shown in Figure 7(e). (a) 3D view. (b) projection onto the xy plane.

These critical points are both saddles (of indices 1 and 2).

Fact 4 *There are at most two critical points (both saddles) in F.*

We next consider the restriction of F to an axis aligned parallelepiped P and mark its edges with the direction of increasing F. The restriction of F to any face of P is the bilinear interpolant discussed in the previous section, therefore facts 4 and 3 imply that one can have maxima of $F|_P$ only at its vertices. Moreover, each face of P can have only two maxima so that the greatest number of maxima of $F|_P$ is four. Figure 7 shows the five distinct types of split trees that can be built with up to four maxima. We show in the following that the last type is not consistent with the topology of the trilinear interpolant.

Fact 5 *The split tree of $F|_P$ cannot have the topology of Figure 7(e).*

Proof: Assume that the tree of Figure 7(e) is a valid split tree for some $F|_P$ with maxima $M1$, $M2$, $M3$ and $M4$. This means that there exist an isovalue w such that the region of P with F greater than w is partitioned into two connected components $R1$ (containing $M1$ and $M2$) and $R2$ (containing $M3$ and $M4$), as shown in Figure 8(a). Since $R1$ is connected we can find a line l_1 that connects $M1$ to $M2$ within $R1$. Similarly we find a line l_2 that connects $M3$ to $M4$ within $R2$.

Let's call $S1$ the front square containing the maxima of $R1$, and $S2$ the back square containing the maxima of $R2$ ($S1$ and $S2$ must be opposite faces of P). We assume, without loss of generality, that $S1$ and $S2$ are orthogonal to the z axis. We consider the parallel projection P along the z axis, onto the xy plane. The images l'_1, l'_2 of l_1, l_2 must intersect in P' (projection of P) because they connect the two pairs of vertices. Their intersection point $r' = l'_1 \cap l'_2$ is the image of a ray r that is parallel to the axis z and that intersects both l_1 and l_2 within P. By construction we have that $F > w$ for $q_1 = r \cap l_1$ and for $q_2 = r \cap l_1$. Moreover, since $R1$ is not connected with $R2$, there must be a point q on r, between q_1 and q_2, where $F < w$. Along r the value of F first decreases from $F(q_1)$ to $F(q)$, and then increases from $F(q)$ to $F(q_2)$. But in a trilinear function the value of F must be monotonic along any line parallel to an orthogonal axis. Thus we have a contradiction, since we have shown that F is not monotonic along r, which is parallel to the z axis. ◇

In conclusion we can state the following:

Theorem 1 *The topology of the split tree of $F|_P$ is completely determined by the count of its local maxima.*

The important practical consequence of this theorem is that we can precompute four templates of split trees and for each element

in the mesh we select the appropriate template simply from the orientation of the edges. Simple numerical computations allow one to determine the specific values of the saddles where the merge occurs.